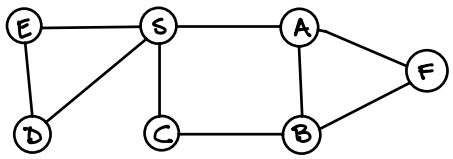
Problem Given a graph G and pair of vertices v and w, what is the **shortest path** from v to w in G?

[Path = sequence of vertices]

First scenario G is unweighted. That is, all edges have distance/cost one.

Example



Shortest path between D and F? D→S→A→F (length = 3)
Shortest path between C and A? C→S→A (length = 2)
Shortest path between E and B? E→S→A→B (length = 3)

Idea Use a modified version of BFS to keep track of distances.

Shortest-Path-BFS

Input Graph $G=(V,E)$ (can be directed or undirected, in adjacency list or adjacency matrix)

A vertex s

Output Distance to all vertices reachable from s

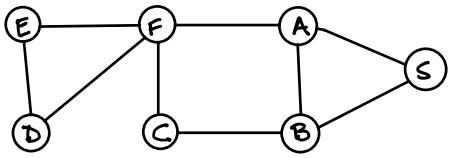
```
dist = new int [|V|]

for each v ∈ V
    dist[v] = ∞
dist[s] = 0

Q.add(s);

while Q ≠ empty
    v = eject(Q)
    for all edges (v,w) ∈ E
        if dist[w] == ∞
            Q.add(w).
            dist[w] = dist[v] + 1.
```

Example



Queue (Q) **Distances (D)**

	A	B	C	D	E	F	S
S	∞	∞	∞	∞	∞	∞	0
A	1	∞	∞	∞	∞	∞	0
A, B	1	1	∞	∞	∞	∞	0
B, F	1	1	∞	∞	∞	2	0
F, C	1	1	2	∞	∞	2	0
F, C, D	1	1	2	3	∞	2	0
C, D, E	1	1	2	3	3	2	0
D, E	1	1	2	3	3	2	0
E	1	1	2	3	3	2	0
∅	1	1	2	3	3	2	0

Second Scenario: Shortest path on weighted graphs.

⇒ So far, we have been treating every edge as having the same weight (1). (That is, all our graphs so far have been **unweighted**)

⇒ But in many applications (perhaps most), having weights on edges is useful.

Example Shortest route in a map: edges between different intersections could have different lengths or weights.

⇒ The edge weights could represent distances, but also money, or cost, or time, etc.

Goal Algorithm for finding shortest paths on weighted graphs

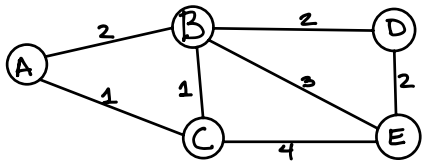
Idea: **Adapt BFS.** The resulting algorithm is known as **Dijkstra's algorithm.**

⇒ Let $G=(V,E)$ be a graph with a weight l_e on each edge $e \in E$

⇒ Suppose l_e is a **positive integer first.**

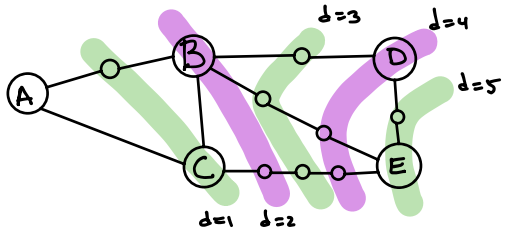
Example:

$G =$



Idea: Add dummy nodes so that every edge weight is one. Then, run BFS.

$\hat{G} =$



From A,

dist[B] = 2

dist[C] = 1, etc.

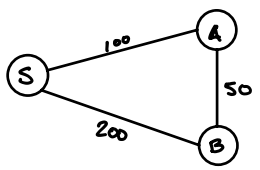
What is the problem with this approach?

(1) Just constructing $\hat{G}$ takes exponential time on the size of the input.

(2) What about non-integer weights? Negative weights?

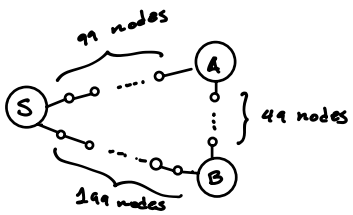
Let us consider a motivating example

$G =$



Add dummy nodes

$\hat{G} =$



⇒ Run BFS from s in $\hat{G}$, and suppose that every second we discover one new layer

⇒ For the first 99 seconds, nothing happens. That is, we are computing distances to dummy nodes we don't really care about.

⇒ Therefore, we could go and take a nap and wake up after 99 seconds when we get to the first real vertex.

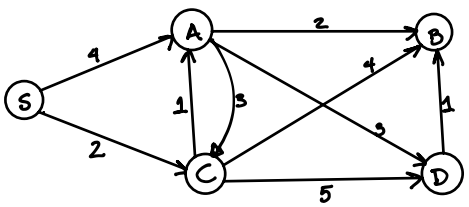
⇒ We can simulate this idea with a system of alarm clocks, where each vertex has one clock. Initially, all the alarms are set to ∞, except the one for the source vertex s that is set to 0.

⇒ The alarm for S rings at 0 and this time we set the alarm for A and B to 100 and 200, respectively.

⇒ These are estimated times (upper bounds actually) for arriving at A and B

⇒ At time 100, the clock for A rings, and we realize that we can set B's alarm to 150.

Let's consider this approach in a more sophisticated example



Alarms

S	A	B	C	D	TIME
0	∞	∞	∞	∞	(initial)
0	4	∞	2	∞	0
0	3	6	2	7	2
0	3	5	2	6	3
0	3	5	2	6	5
0	3	5	2	6	6

What we have described, is Dijstra's algorithm: Given a graph and a source vertex s, Dijkstra's algorithm find the shortest path from s to any other vertex.

⇒ We will now provide an implementation

⇒ Later we will prove the correctness of the algorithm and will discuss faster implementations

Dijkstra

Input A graph $G=(V,E)$ and a source vertex S

Output All shortest paths from s to any other vertex

```
dist = new int [|V|]      // These are the alarm clocks

for each v∈V
    dist[v]=∞
dist[s]=0                 } Initialization.

Repeat |V| times:
    ⊗ Find the vertex v whose alarm clock has not rang yet
      with the smallest alarm time.

    for each (v,w) ∈E
        if  dist[w] > dist[v] + l(v,w)
            dist[w]= dist[v] +l(v,w)
```

Running Time (with naive implementation) – $O(|V|^2)$

⇒ To do better, we need to optimize ⊛, and we'll do it later.

Proof of Correctness

Lemma If at time T the alarm clock for vertex v rings for the first time, the the shortest path from s to v has weight T.

Proof Let R be the set of vertices whose alarm has already rang.

It suffices to prove that at any time, for every vertex $w \in R$, the time at which the alarm for w rang is the shortest path from s to w.

Initially $R = \{\emptyset\}$ and at time $T=0$, s is added.

Then, $R = \{s\}$ and $dist[s] = 0$, which is correct.

We show next that the property is preserved every time we add a vertex to R.

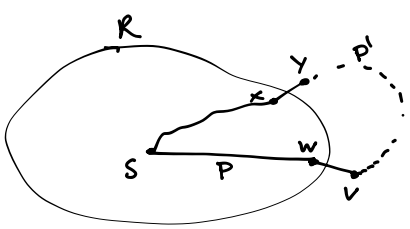
Suppose the alarm of v goes off first. This means that:

(1) $dist[v] = \min_{z \notin R} dist[z]$.

(2) v is added to R

(3) $dist[v] = dist[w] + l(w,v)$ for some $w \in R$ (since $l(w,v) \geq 0$)

Here is the situation:



Consider the path $P = s \rightsquigarrow w \rightarrow v$

We can show that P is a shortest path from s to v.

Suppose (by contradiction) that this is not the case.

Then, ∃ a path P' from s to v with smaller weight.

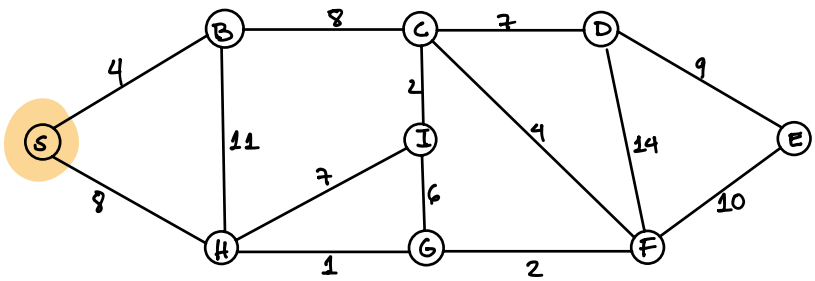
Let x be the last vertex from R in P' and y the next vertex.

Note that,

length of $P' \geq$ dist[x] + $l(x,y) \geq$ dist[y] $\geq$ dist[v] = length of P

Which is a contradiction and the proof is complete.

One more Example



DIST	S	B	C	D	E	F	G	H	I
	0	∞	∞	∞	∞	∞	∞	∞	∞
0		4	∞	∞	∞	∞	∞	8	∞
0	4		12	∞	∞	∞	∞	8	∞
0	4		12	∞	∞	∞	9	8	15
0	4		12	∞	∞	11	9	8	15
0	4		12	25	21	11	9	8	15
0	4	12		19	21	11	9	8	14
0	4	12		19	21	11	9	8	14
0	4	12		19	21	11	9	8	14
0	4	12		19	21	11	9	8	14

Elements in R shown in Blue

Speeding up our implementation of Dijkstra.

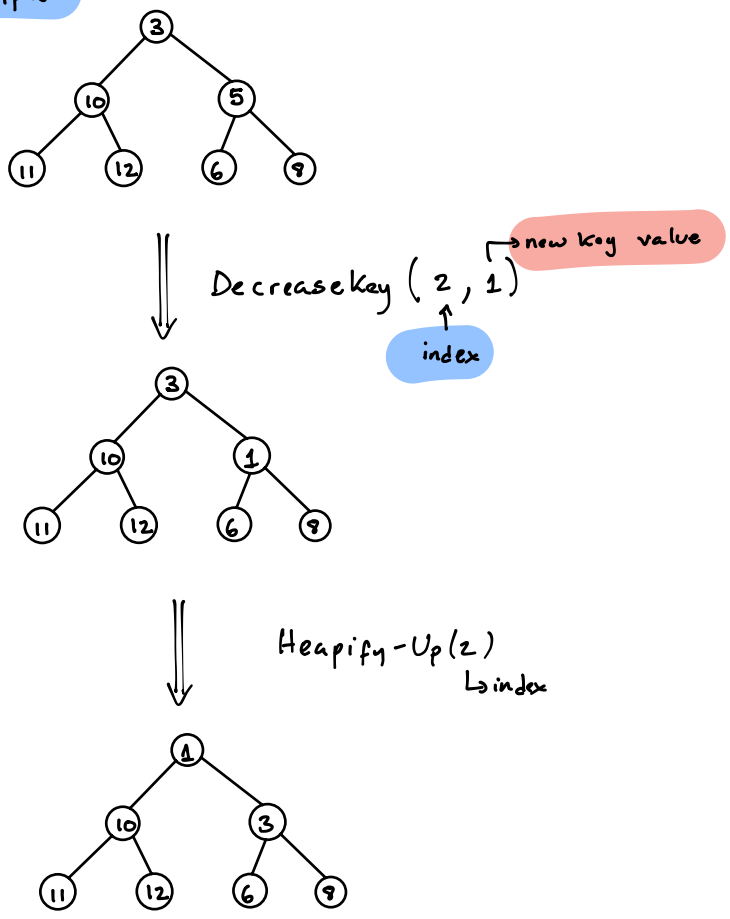
Note that we need to find, in each iteration of the loop, the vertex with the minimum alarm time.

⇒ This can be done using a **min Heap** that maintains pairs of [alarm time, vertex]

⇒ But note that the alarm time of the vertices may be updated sometimes, so we need to add this operation to our heaps.

⇒ The operation is called **Decrease key** and decreases the key of a given element and heapifies up to maintain the heap property.

Example



⇒ Running time of DecreaseKey : $O(\log n)$ (in a heap with n elements)

⇒ Min heaps that support DecreaseKey give an implementation of the abstract data structure Priority Queue.

(Implementation using Priority Queue in next page)

Dijkstra - PriorityQueue

Input A graph $G=(V,E)$ and a source vertex S

Output All shortest paths from S to any other vertex

```
dist = new int [|V|]
prev = new int [|V|]   ⟶ to reconstruct the path
for each v∈V
    dist[v]=∞
    pre[v]=null

dist[s]=0

Build a Heap H using {dist[v], v} as {key, value} pairs

Our implementation of Heap will also keep track of the position of
each vertex v in the Heap. So that position[v] returns the index
of v in the heap. This is possible by making the swap operation
update the array position after each swap.

Repeat |V| times:
    v = GetMin(H)
    Remove(H, 0)

    for each (v,w)∈E
        if dist[w] > dist[v] + l(v,w)
            dist[w] = dist[v] + l(v,w)
            DecreaseKey(H, position[w], dist[w])
            pre[w] = v
```

Running Time: $O((|V|+|E|)\log|V|)$

Can we do better? With more fancy heaps (known as **Fibonacci Heaps**) we can achieve: $O(|V|\cdot\log|V|+|E|)$

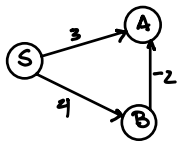
but this speed up is only interesting theoretically.

Final Remarks:

- Dijkstra's require all edge weights to be **non-negative**
- The actual path can also be recovered using the prev array.

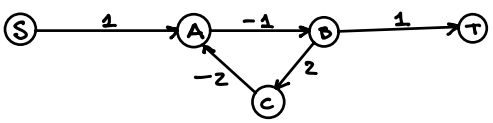
Scenario 3: Shortest path with negative weights

⇒ With negative edges, Dijkstra's algorithm does not work. For example, consider:



Dijkstra's algorithm will claim that S → A is the best path from S to A, but S → B → A has smaller weight.

⇒ We can allow negative edges but not negative cycles, since in this case the shortest path problem is not well-defined. For example:



The shortest path from S to + has $-\infty$ weight!

Bellman-Ford Algorithm

→ Shortest path in the precence of negative weights.

⇒ In Dijkstra's algorithm, we maintain an array of distances (alarm times).

⇒ The values on the array are either overestimates or exactly correct values for the shortest paths.

⇒ The only way an entry from dist[·] is reduce is by performing the "Update operation".

```
Update ((v,w) ∈ E)
    if dist[w] > dist[v] + l(v,w)
        dist[w] = dist[v] + l(v,w)
```

⇒ Dijkstra's algorithm can be seen as a clever sequence of Update operations.

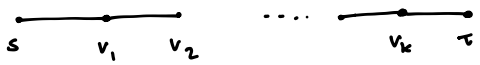
⇒ This sequence of Update operations does not work with negative edges, but what about other sequences of updates?

Two key properties of the Update operation:

(1) If dist[v] has the correct value for the shortest path from source vertex s to v, and the last edge in the shortest path from s to w is (v,w), then if we call Update(v,w), dist[w] will contain the length of the shortest path from s to w.

(2) It will never make dist[w] too small for any $w \in V$, since it only contains lengths through valid paths. This means that it is a **safe operation** in the sense that we can Update as many times as we like.

Now, consider a shortest path from s to t:



⇒ There are at most $|V|-1$ edges in the path

⇒ Suppose we call Update first in (s,v_1), then (v_1,v_2), (v_2,v_3),... in that order. At the end of the process we would have discovered the shortest path (see property (1) above)

⇒ What happens if we insert additional updates?
Nothing! by property (2).

⇒ So, if we knew shortest path, we are done. But if not?

Solution Call Update for all the edges $|V|-1$ times.

Bellman-Ford

Input: Graph $G=(V,E)$ with weights $\ell(e)$ for $e \in E$
Vertex s

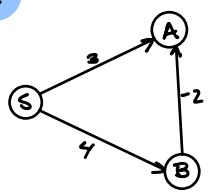
Output: Shortest path from s to any other vertex

```
for all w∈V
    dist[w]=∞
dist[s]=0

Repeat |V|-1 times
    for each e∈E
        Update(e)
```

Running Time $O(|V| \cdot |E|)$

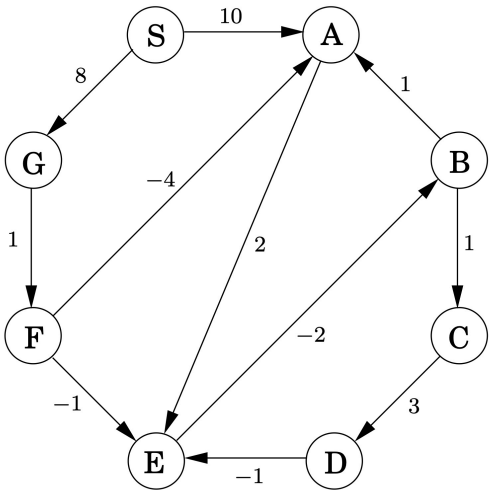
Example



S	A	B	
0	∞	∞	(S,A), (S,B), (B,A)
0	3	∞	Update(S,A)
0	3	4	Update(S,B)
0	2	4	Update(B,A)

Bellman-Ford will call Update three more times but nothing will get updated.

Example



Edges: (S, A), (B, A), (B, C), (C, D), (D, E), (F, E), (G, F), (S, G), (F, A), (A, E), (E, B)

S	A	B	C	D	E	F	G	Iteration
0	∞	∞	∞	∞	∞	∞	∞	(initial)
0	10	10	∞	∞	12	∞	8	1
0	5	5	11	14	7	9	8	2
0	5	5	6	9	7	9	8	3
0	5	5	6	9	7	9	8	4

⇒ No updates happen in the fourth iteration, so we stop at that time.

Scenario 4: All pairs shortest path in directed graphs

⇒ If run Bellman-Ford from each vertex, to obtain the shortest paths from all vertices, the running time is $O(|V|^2 \cdot |E|)$ which can be $O(|V|^4)$ from the graph is dense

⇒ A better algorithm is Floyd-Warshall algorithm

Floyd-Warshall algorithm

⇒ Denote the vertices with labels $\{1, 2, \ldots, n\}$ for simplicity.

⇒ Let $P(i, j, k)$ denote the shortest path from i to j but only using vertices $\text{fin } \{1, 2, \ldots, k\}$

Goal: Find $P(i,j,n)$ $\forall i,j \in \{1,\dots,n\}$

Idea: Use recursion but in a clever/organized way.

(1) $P(i,j,k) \leq P(i,j,k-1)$

(2) If $P(i,j,k) < P(i,j,k-1)$ then $\exists$ a path from i to j using vertices from $\{1,\dots,k-1\}$ and k that is shorter than only using from $\{1,\dots,k-1\}$

(3) Such path will go $i \rightsquigarrow k \rightsquigarrow j$ and only using vertices from $\{1,\dots,k-1\}$ to go from i to k and the from k to j.

(4) So we arrived at the following:

$$P(i,j,k) = \min\{P(i,j,k-1),\ P(i,k,k-1)+P(k,j,k-1)\}$$

(*) So to compute $P(i,j,k)$, it suffices to compute $P(\cdot,\cdot,k-1)$.

Base case: $P(i,j,0) = \begin{cases} \ell(i,j) & \text{if } (i,j)\in E \\ 0 & \text{o.w.} \end{cases}$

$\Rightarrow$ The Floyd-Warshall Algorithm works by computing $P(i,j,1)$ for all pairs (i,j), then $P(i,j,2)$ for all pairs and so on.

Floyd-Warshall

Input: $G=(V,E)$ $\ell: E \to \mathbb{R}$

Output: Shortest path between all pairs.

```
dist = new int [|V|][|V|]
for each (i,j) ⇒ dist[i][j] = { 0 if i=j
                              { ℓ(i,j) if (i,j)∈E
                              { ∞ otherwise
for k=1 to n
    for i=1 to n
        for j=1 to n
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] = dist[i][k] + dist[k][j]
```

Running Time: $O(|V|^3)$

We have covered four different algorithms for shortest paths but there are many more that work better in some situations.

Here are some other algorithms and scenarios with their respective running time.

Scenario	Algorithm	Running Time
Undirected, positive weights	Floyd-Warshall	$O(\|V\|^3)$
directed, no negative cycles	Johnson - Dijkstra	$O(\|E\|\|V\| + \|V\|^2 \log\|V\|)$
directed, no negative cycles	Pettie (2004)	$O(\|E\|\|V\| + \|V\|^2 \log\log\|V\|)$

Many more!!